CS3025 Compiladores

Semana 3:
Flex (Introduccion)
un generador de analizadores lexicos

28 Agosto 2023

Igor Siveroni

Semana 3

Lunes 27 Agosto:
- Flex: teoría + laboratorio

Martes 28 Agosto:
- Analisis Sintactico: Parsers
- Parsers LL(1)
- Recursive Descent / Descenso Recursivo

Viernes 1 Septiembre
- Laboratorio: Parsers LL(1)

Lectura: Capitulo 3, A. Appel

Analisis Lexico

El objetivo del análisis léxico es el de leer una secuencia de caracteres (programa fuente) y producir como output una secuencia de tokens

Hemos aprendido lo siguiente:

- El análisis léxico reconoce cadenas descritas por una serie de patrones y genera tokens para (casi todas) las cadenas reconocidas e.g. no genera tokens para espacios en blanco.
- Los patrones son descritos usando Expresiones Regulares.
- Toda expresión regular genera (por lo menos) un automata finito determinístico que puede ser usado una verificar si una cadena pertenece al lenguaje descrito por la expresión regular.
- Dada la lista de patrones, implementaciones clásicas de scanners codifican el autómata resultante de la unión de los autómatas equivalentes a cada patron.

Analisis Lexico: Generacion automatica de scanners

El proceso de implementar scanners ‘a mano’ puede ser largo y vulnerable a errores.

Hemos visto (al menos en teoría) que el proceso de generar automatas finitos deteministicos a partir de expresiones regulares esta regido por una serie de algoritmos.

Si tenemos los algoritmos, podemos escribir un programa que automitiza este proceso.

Eso es precisamente lo que hace un Generador Automatico de Analizadores Lexicos.

En el curso usaremos **FLEX**

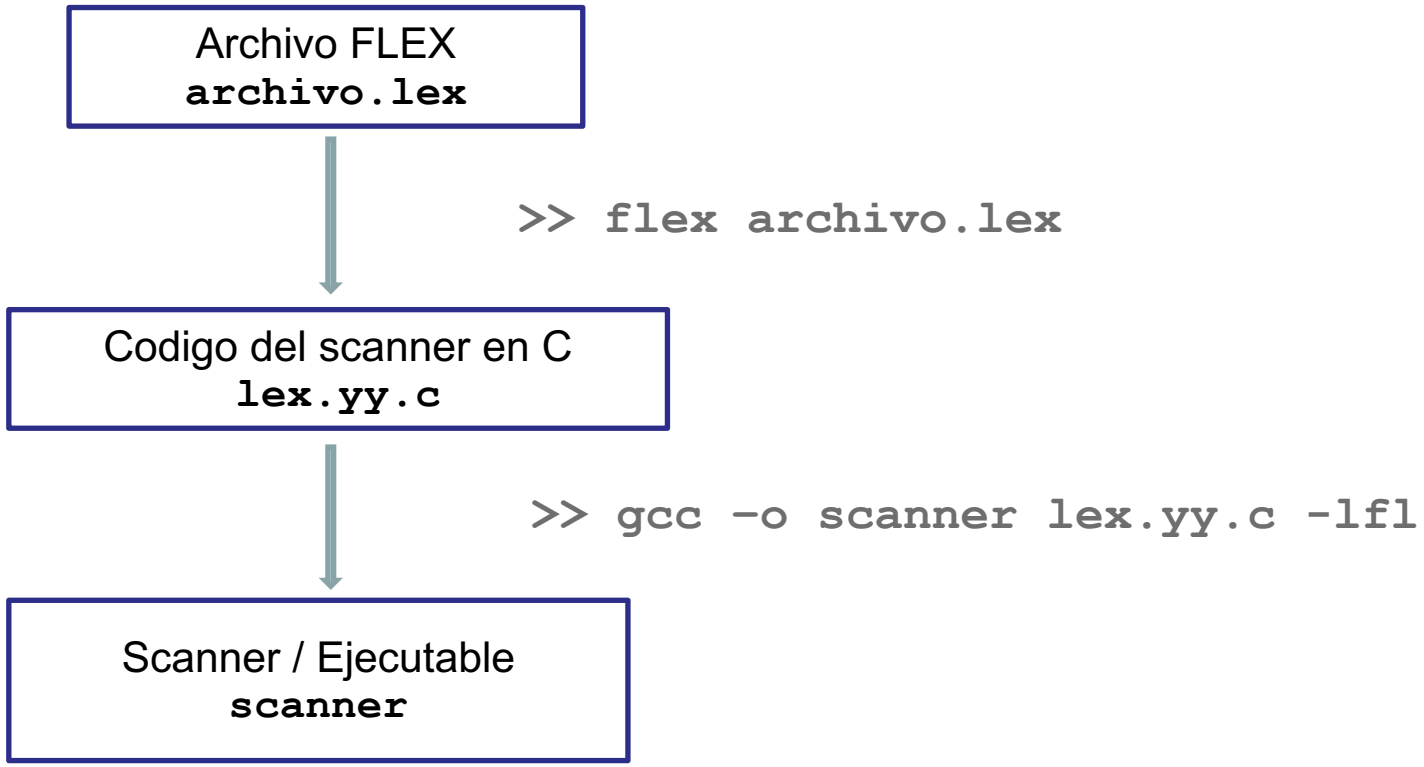
FLEX

- Lex es un programa que genera analizadores léxicos.
- Lex toma como entrada un archivo de texto que contiene reglas que asocian expresiones regulares con acciones.

- La versión más popular de Lex se conoce como Flex.

- Flex tiene como salida un fichero fuente en C, lex.yy.c, el cual se compila para producir un ejecutable.

FLEX: Fast Lexical Analyzer Generator



Expresiones Regulares en Flex I

Patrón	Expresión
x	empareja el caracter x
$[xyz]$	clase de caracteres
$[a-z]$	clase de caracteres con rango
$[\wedge A-Z]$	clase de caracteres negada
$r*$	cero o más r
$r+$	una o más r
rs	concatenación
$r\|s$	unión

Ejemplo: `[^a]` todos los caracteres con excepcion de la `a`

Expresiones Regulares en Flex II

Patrón	Expresión
$r?$	cero o una r
$r\{3,5\}$	de 3 a 5 r's
$r\{4,\}$	4 o más r's
$r\{4\}$	4 r's
$\backslash n$	salto de linea
$\backslash t$	espacio
.	cualquier caracter distinto al salto de linea
{nombre}	expansión de definición

Ejemplo

Patrón	Lexemas
$a\|b$	a,b
$[ab]$	a,b
ab	ab
$(ab)^*$	ϵ, ab,abab,etc
$ab\{1,3\}$	ab,abb,abbb
$[a-z]$	a,b,c,...z
$[\wedge a]$	bccd, dlc, xyz, etc
$[a-zA-Z]$	a,b,c,...,z,A,B,C,...,Z
$[a-zA-Z0-9]$	a,b,c,...,z,A,B,C,...,Z,0,1,2,...,9

Ejemplo

Patrón	Token
$[0-9]$	Digito
$[a-zA-Z]$	Letra
$[" + "]$	Suma
$" >= "$	Mayor o igual
$["."]$	Punto
$[-\_]$	Guiones
$\{Letra\}(\{Letra\}\|\{Digito\}\|\{Guiones\})^*$	Identificador
$(-?[1-9][0-9])\|0$	Entero
$\{Digito\}\{0,8\}\{Punto\}\{Digito\}\{1,8\}$	Float
$"if"\|"IF"$	IF

Variables Globales y Funciones Predefinidas

Variables globales:

- `yytext`: cadena que contiene el texto reconocido (char*)
- `yyleng`: longitud de yytext (int)
- `yyin`: puntero al fichero de entrada
- `yyout`: puntero al fichero de salida

Funciones predefinidas

- `yylex`(): Llama al analizador lexico generado por flex.
- `yymore`(): indica a flex que anada el siguiente componente lexico al componente lexico actual
- `yywrap`(): se ejecuta cuando el analizador lexico encuentra el fin de fichero.
- `yyless`() retiene los primeros n caracteres de yytext y resto al dispositivo de lectura

Archivo FLEX

En archivo de entrada Flex se compone de tres partes: una colección de definiciones, una colección de reglas y una colección de rutinas auxiliares. Las tres secciones están separadas por signos de porcentaje dobles que aparecen en líneas separadas comenzando en la primera columna

```
// definiciones

%%

//reglas

%%

// rutinas auxiliares
```

Archivo FLEX: definiciones

La sección dedicada a definiciones contiene:

- código en C escrito entre los símbolos `%{` y `%}`, y
- una serie de definiciones de la forma `nombre patron`.

```
%{
        // código C
%}
nombre1 patron1
nombre2 patron2
…

%%      // fin de definiciones
```

Archivo FLEX: definiciones - ejemplo

```
%{
int num_lines = 0, num_chars = 0;   // global vars
%}
digito [0-9]
numero {digito}+
%%
```

- El código en C puede declarar variables globales, incluir directivas del preprocesador e.g. `#include`, etc.
- Importante: es necesario usar { y } para hacer referencia a definiciones e.g. `{digito}`.

Archivo FLEX: Reglas

La segunda parte de un archivo Flex contiene una serie de reglas de la forma `patron accion`.

- Las reglas deberan ser NO indentadas.
- Las acciones son codigo en C/C++ ha ejecutarse cada vez que el patron de la regla es satisfecho.
- Se recomienda incluir las acciones entre { … }. De esta manera se pueden incluir acciones que abarquen mas de una linea.
- Solo una regla es activada a la vez.

```
%%
{numero}     { printf("Numero: %s\n", yytext);
               números++;   }
aa |
bb            printf("aa o bb\n");
%%
```

Archivo FLEX: Funciones Auxiliares

La tercera sección incluye código auxiliar en C. Este código es copiado verbatim al analizador lexico generado.
Esta parte es opcional.

Por ejemplo, si se planea incluir la función main() para asi crear un ejecutable del analizador lexico, tendríamos algo asi:

```
%%
int main() {
  yylex();
  printf( "# of lines = %d, # of chars = %d\n",
          num_lines, num_chars );
}
```

Ejemplo: Expresiones aritmeticas

```
digito      [0-9]
numero      {digito}+
ws          [ \t]+
%%
{ws}         { // consume caracteres en blanco
"("          { printf("LPAREN\n"); }
…
"+"          { printf("PLUS\n"); }
{numero}     { printf("NUM");  }
.            { printf("error: %s\n",  yytext); }
%%
```

Importante: Agregar la regla con un punto al final para capturar errores. El punto denota cualquier carácter (menos cambio de línea) – funciona como un default en un switch.

Ejemplo: Identificadores y palabras reservadas

```
ID     [a-z][a-z0-9]+
…
%%
if | then | else   {  printf(“reservada %s\n”, yytext) }
{ID}               { printf(“NUM”);  }
.                  { printf("error: %s\n",  yytext); }
%%
```

En la especificación de arriba las palabras reservadas también satisfacen el patron para ID.
La regla en flex es: si mas de una regla hace el match con la cadena, se escoge la regla que aparece primero en la especificación.

Conclusion: Colocar las reglas para palabras reservadas arriba de la IDs.

Flex: Comentario Ejemplos

- Nuestros ejemplos, por ahora, reconocen cadenas e imprimen información relevante a los patrones y textos emparejados.
- Las acciones en Flex pueden realizar operaciones mucho mas complejas e.g. manipulación de texto.
- En el contexto de un compilador, las acciones generan Tokens, los cuales son leídos por el analizador sintáctico (parser).
- Si estamos usando Flex, posiblemente estaremos usando Bison para generar el analizador sintactico.
- Flex y Bison están diseñados para trabajar juntos.
- Veremos esta interacción en detalle luego de ver Análisis Sintáctico.

Flex: Uso

- Flex genera el archivo lex.yy.c por defecto. Es posible especificar el nombre del fichero en C con la opción –o:

```
>> flex –o exp.lex.c exp.lex
>> gcc –o exp exp.lex.c -lfl
>> ./exp.                          --- para ejecutar scanner
>> ./exp <  input.txt.     -- redirecciona stardard input
```

- Flex genera archivos en C. Es posible insertar código en C++ en el archivo Flex y luego compilar el programa generado como un archivo C++:

```
>> flex –o exp_cpp.lex.c exp_cpp.lex
>> g++ -o  exp_cpp exp_cpp.lex.c     -- compilador C++
```

  Esto debería funcionar con Bison también.
  Importante por que vamos a necesitar insertar acciones que llaman a constructores C++.